Diving Into The Ethereum Virtual Machine

zh

Follow

Qtum

--

9

Listen

Share

Solidity offers many high-level language abstractions, but these features make it hard to understand what's really going on when my program is running. Reading the Solidity documentation still left me confused over very basic things.

What are the differences between string, bytes32, byte[], bytes?

- Which one do I use, when?
- What's happening when I cast a string to bytes? Can I cast to byte[]?
- How much do they cost?

How are mappings stored by the EVM?

- Why can't I delete a mapping?
- Can I have mappings of mappings? (Yes, but how does that work?)
- Why is there storage mapping, but no memory mapping?

How does a compiled contract look to the EVM?

- How is a contract created?
- What is a constructor, really?
- What is the fallback function?

I think it's a good investment to learn how a high-level language like Solidity runs on the Ethereum VM (EVM). For couple of reasons.

1. Solidity is not the last word.

Better EVM languages are coming. (Pretty please?)

1. The EVM is a database engine.

To understand how smart contracts work in any EVM language, you have to understand how data is organized, stored, and manipulated.

1. Know-how to be a contributor.

The Ethereum toolchain is still very early. Knowing the EVM well would help you make awesome tools for yourself and others.

1. Intellectual challenge.

EVM gives you a good excuse to play at the intersection of cryptography, data structure, and programming language design.

In a series of articles, I'd like to deconstruct simple Solidity contracts in order to understand how it works as EVM bytecode.

An outline of what I hope to learn and write about:

- The basics of EVM bytecode.
- How different types (mappings, arrays) are represented.
- What is going on when a new contract is created.

- What is going on when a method is called.
- How the ABI bridges different EVM languages.

My final goal is to be able to understand a compiled Solidity contract in its entirety. Let's start by reading some basic EVM bytecode!

This table of EVM Instruction Set would be a helpful reference.

A Simple Contract

Our first contract has a constructor and a state variable:

Compile this contract with solc

:

The number 6060604052...

is bytecode that the EVM actually runs.

In Baby Steps

Half of the compiled assembly is boilerplate that's similar across most Solidity programs. We'll look at those later. For now, let's examine the unique part of our contract, the humble storage variable assignment:

This assignment is represented by the bytecode 6001600081905550

. Let's break it up into one instruction per line:

The EVM is basically a loop that execute each instruction from top to bottom. Let's annotate the assembly code (indented under the label tag_2

) with the corresponding bytecode to better see how they are associated:

Note that 0x1

in the assembly code is actually a shorthand for push(0x1)

. This instruction pushes the number 1 onto the stack.

It still hard to grok what's going on just staring at it. Don't worry though, it's simple to simulate the EVM line by line.

Simulating The EVM

The EVM is a stack machine. Instructions might use values on the stack as arguments, and push values onto the stack as results. Let's consider the operation add

.

Assume that there are two values on the stack:

When the EVM sees add

, it adds the top 2 items together, and pushes the answer back onto the stack, resulting in:

In what follows, we'll notate the stack with []

:

And notate the contract storage with {}

:

Let's now look at some real bytecode. We'll simulate the bytecode sequence 6001600081905550

as EVM would, and print out the machine state after each instruction:

The end. The stack is empty, and there's one item in storage.

What's worth noting is that Solidity had decided to store the state variable uint256 a

at the position 0x0

. It's perfectly possible for other languages to choose to store the state variable elsewhere.

In pseudocode, what the EVM does for 6001600081905550

is essentially:

Looking carefully, you'd see that the dup2, swap1, pop are superfluous. The assembly code could be simpler:

You could try to simulate the above 3 instructions, and satisfy yourself that they indeed result in the same machine state:

Two Storage Variables

Let's add one extra storage variable of the same type:

Compile, focusing on tag_2

:

The assembly in pseudocode:

What we learn here is that the two storage variables are positioned one after the other, with a

in position 0x0

and b

in position 0x1

.

Storage Packing

Each slot storage can store 32 bytes. It'd be wasteful to use all 32 bytes if a variable only needs 16 bytes. Solidity optimizes for storage efficiency by packing two smaller data types into one storage slot if possible.

Let's change a

and b

so they are only 16 bytes each:

Compile the contract:

The generated assembly is now more complex:

The above assembly code packs these two variables together in one storage position (0x0

), like this:

The reason to pack is because the most expensive operations by far are storage usage:

- sstore

costs 20000 gas for first write to a new position.

- sstore

costs 5000 gas for subsequent writes to an existing position.

- sload

costs 500 gas.

- Most instructions costs 3~10 gases.

By using the same storage position, Solidity pays 5000 for the second store variable instead of 20000, saving us 15000 in gas.

More Optimization

Instead of storing a

and b

with two separate sstore

instructions, it should be possible to pack the two 128 bits numbers together in memory, then store them using just one sstore

, saving an additional 5000 gas.

You can ask Solidity to make this optimization by turning on the optimize

flag:

Which produces assembly code that uses just one sload and one sstore:

The bytecode is:

And formatting the bytecode to one instruction per line:

There are four magic values used in the assembly code:

- 0x1 (16 bytes), using lower 16 bytes
- 0x2 (16 bytes), using higher 16bytes
- not(sub(exp(0x2, 0x80), 0x1))
- sub(exp(0x2, 0x80), 0x1)

The code does some bits-shuffling with these values to arrive at the desired result:

Finally, this 32bytes value is stored at position 0x0

.

Gas Usage

60008054700200000000000000000000000000000000

6001608060020a031990911660011760016080600 20a0316179055

Notice that 0x20000000000000000000000000000000

is embedded in the bytecode. But the compiler could've also chosen to calculate the value with the instructions exp(0x2, 0x81)

, which results in shorter bytecode sequence.

But it turns out that 0x20000000000000000000000000000000

is a cheaper than exp(0x2, 0x81)

. Let's look at the gas fees involved:

- 4 gas paid for every zero byte of data or code for a transaction.
- 68 gas for every non-zero byte of data or code for a transaction.

Let's compare how much either representation costs in gas.

- The bytecode 0x20000000000000000000000000000000

. It has many zeroes, which are cheap.

(1 * 68) + (16 * 4) = 196.

- The bytecode 608160020a

. Shorter, but no zeroes.

5 * 68 = 340.

The longer sequence with more zeroes is actually cheaper!

Summary

An EVM compiler doesn’t exactly optimize for bytecode size or speed or memory efficiency. Instead, it optimizes for gas usage, which is an layer of indirection that incentivizes the sort of calculation that the Ethereum blockchain can do efficiently.

We’ve seen some quirky aspects of the EVM:

- EVM is a 256bit machine. It is most natural to manipulate data in chunks of 32 bytes.
- Persistent storage is quite expensive.
- The Solidity compiler makes interesting choices in order to minimize gas usage.

Gas costs are set somewhat arbitrarily, and could well change in the future. As costs change, compilers would make different choices.